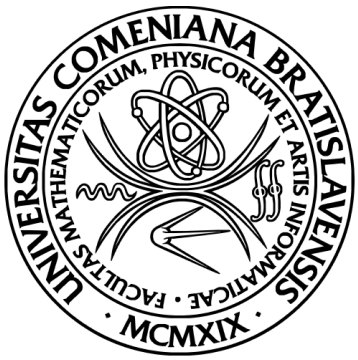
UNIVERZITA KOMENSKÉHO V BRATISLAVE FAKULTA MATEMATIKY,
FYZIKY A INFORMATIKY



ALGORITMY RIADENIA HUMANOIDNÉHO ROBOTA

Bakalárska práca

2020 TOMÁŠ KOSEC

UNIVERZITA KOMENSKÉHO V BRATISLAVE FAKULTA MATEMATIKY,
FYZIKY A INFORMATIKY

ALGORITMY RIADENIA HUMANOIDNÉHO ROBOTA

BAKALÁRSKA PRÁCA

Študijný program: Aplikovaná informatika
Študijný odbor: 2511 Aplikovaná informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: Mgr. Pavel Petrovič, PhD.

Bratislava 2020 Tomáš Kosec

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Tomáš Kosec
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Algoritmy riadenia humanoidného robota
Algorithms for Humanoid Robot Control

Anotácia: Robotické zariadenia sa postupne stávajú súčasťou každodenného života. Nie sú žiadne špeciálne dôvody pre stavbu a štúdium humanoidných robotov, s výnimkou toho, že ľudia budú strojom, ktoré viac pripomínajú živé stvorenia, pravdepodobne viac dôverovať a toho, že naše prostredie je v zmysle možností pohybu a manipulácie prispôsobené pre vzpriamených dvojnožcov. Robot Lilli je humanoidný robot s 25 stupňami voľnosti, navrhnutý Perom Salkowitschom, ktorý ho prvýkrát prezentoval na výstave Maker Fair vo Viedni v roku 2018. Následne jeden exemplár študenti v našej robotickej skupine poskladali a Lilli dostal riadiacu elektroniku a program na vytváranie jednoduchých choreografií a prvé pokusy o balancovanie. Neskôr bol vybavený 3D hĺbkovým videním a použitý v projekte na robotickom kurze. Je však potrebné vynaložiť veľa úsilia, aby bol Lilli schopný pohybov a manipulácie v prostredí. Cieľom tejto práce je pokročiť v tomto projekte, konkrétne: navrhnúť, implementovať a overiť algoritmy pre pohyb a manipuláciu v prostredí robota Lilli, porovnať ich vhodnosť a zvážiť použitie automatických metód na ladenie parametrov algoritmov a zamerať sa aj na spôsoby interakcie človeka s robotom.

Literatúra:

Poznámka: Illah Reza Nourbakhsh and Roland Siegwart: Introduction to Autonomous Mobile Robots, MIT Press, 2004.
Howie Choset et al: Principles of Robot Motion, Theory, Algorithms, and Implementations, MIT Press, 2005.

Kľúčové slová: humanoidný robot, Lilli, pohyb a manipulácia robotov

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 15.10.2019

Dátum schválenia: 15.10.2019 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Rád by som poďakoval môjmu školiteľovi, Mgr. Pavlovi Petrovičovi, PhD., za jeho pomoc, podporu a cenné rady pri písaní tejto práce.

Abstrakt

Humanoidní roboti sa každým dňom zdokonaľujú v ľudských činnostiach. Ich vývoj zaznamenal posledné roky veľký rozmach a pokrok. Stávajú sa kompaktnejšími a dokonalejším a už dnes pomáhajú v mnohých oblastiach, kde je nasadenie ľudského života náročné či nebezpečné. Cieľom tejto práce bolo navrhnúť, implementovať a overiť algoritmy pre pohyb a manipuláciu v prostredí humanoidného robota Lilli. Ako programovací jazyk sme zvolili jazyk C++ kvôli jeho rýchlosti a podpore v používaných nástrojoch. Prvým krokom bola detekcia objektov v scéne pred Lilli. Na zaznamenávanie obrazu sme využili kameru ZED mini, ktorá zaznamenala mračno bodov v ktorom sme následne detegovali objekty; na tento úkon sme využili knižnicu Point Cloud Library, ktorá v sebe obsahuje užitočné nástroje na prácu s mračnom bodov a detekciu objektov. Druhým krokom bola navigácia ramena k detegovanému objektu. Túto úlohu sme riešili za pomoci inverznej kinematiky, presnejšie pomocou softvéru Orocos a jeho časti Kinematic and Dynamic. Taktiež sme preskúmali možnosti chôdze a vytvorili pomocnú konštrukciu na riešenie tejto úlohy.

Kľúčové slová: humanoidny robot, Lilli, pohyb a manipulácia robotov, mračno bodov, detekcia objektov, inverzná kinematika

Abstract

Humanoid robots improve in human-like activities every day. In recent years, their development has seen great growth and progress. They are becoming more compact and more perfect, and they are already helping in many areas where the deployment of human life is difficult or dangerous. The aim of this work was to design, implement and verify algorithms for movement and manipulation in the environment of the humanoid robot Lilli. We chose C++ as the programming language due to its speed and support in the tools used. The first step was to detect objects in the scene in front of Lilli. To capture the image, we used the ZED mini camera, which captured point cloud in which we detected objects; for this purpose, we used the Point Cloud Library, which contains useful tools for working with point cloud and detecting objects. The second step was to move the arm towards the detected object. We solved this task by inverse kinematics, more precisely with the help of Orocos software and its Kinematic and Dynamic part. We also explored the possibilities of walking and created an auxiliary structure to solve this problem.

Key words: humanoid robot, Lilli, movement and manipulation of robots, point cloud, object detection, inverse kinematics

Obsah

Úvod

Podľa definície slova (humanoidný) robot, teda robot, ktorý imituje pohyby človeka a má horné a dolné končatiny, môžeme za prvého humanoidného robota považovať *Herbert*-a *Televox*-a vytvoreného pánom Roy-om Wensley-om v roku 1927. Išlo o robota, ktorý bol vytvorený za účelom zdvíhania prijímača a tak uskutočňovať telefonický hovor. Koncom šesťdesiatych rokov sa začal projekt WABOT na Waseda University. V roku 1972 bol dokončený robot WABOT-1, ktorý bol zároveň aj prvým humanoidným robotom v životnej veľkosti, ktorý vedel chodiť, komunikovať s ľuďmi (v japončine), navigovať sa po miestnosti a zdvíhať i premiestňovať objekty. Odhadovalo sa, že WABOT-1 má mentálnu schopnosť jeden a pol roka starého dieťaťa. V roku 1980 sa začalo s vývojom robota WABOT-2, ktorý po dokončení vedel navyše čítať noty a hrať na klavír[19].

Vývoj humanoidných robotov zažil v poslednej dekáde obrovský rozvoj. Humanoidné roboty zanedlho budú súčasťou nášho každodenného života v rôznych oblastiach našej spoločnosti. Ich využite môže byť veľkým prínosom pri rizikových povolaniach či povolaniach, kde dochádza k nadmernej záťaži na ľudský organizmus. Humanoidné roboty sú zároveň vhodné aj na interakciu s človekom, keďže sa naňho podobajú - z psychologického hľadiska si k nim môžu ľudia vybudovať dôvernejší vzťah ako ostatným typom robotov.

Cieľom tejto bakalárskej práce je zdokonaliť humanoidného robota Lilli, ktorý sa nachádza na Fakulte matematiky, fyziky a informatiky. Prvým cieľom je vytvorenie algoritmu na detekciu okolia a objektov v ňom. Na základe detegovaných objektov a ich súradníc využijeme inverznú kinematiku na navigovanie robotického ramena k objektu. Ďalším cieľom je preskúmať možnosti kráčania.

Kapitola 1

Prehľad problematiky

1.1 Humanoidný robot

Za humanoidného robota sa považuje každý robot, ktorý má stavbu tela podobnú tomu ľudskému. Teda má torzo, horné končatiny, dolné končatiny a hlavu, a k pohybu využíva bipédiu, teda pohyb po dvoch zadných končatinách (v našom vnímaní je to chôdza). Za humanoidného robota sa považuje aj robot, ktorý má len torzo, horné končatiny a hlavu, a imituje človeka. Niektorí humanoidní roboti majú prepracovanú hlavu, teda majú oči, ústa, nos a uši. Roboti, ktorí reálne imitujú človeka, hlavne výzorom, sa nazývajú androidi.

Humanoidné roboty sú oblasťou výskumu mnohých vedných disciplín. Pri konštrukcii humanoidného robota je potrebná dokonalá znalosť ľudského tela, teda takýto výskum je prospešný nielen pre robotiku ale i pre ostatné kognitívne vedy so zameraním na ľudské telo a jeho správanie. Pôvodným cieľom humanoidného výskumu však bolo vytváranie ortéz a protéz pre ľudí, tieto znalosti sa však ľahko preniesli aj do robotiky, kde našli veľké uplatnenie.

Humanoidní roboti sú určení prevažne na vykonávanie ľudských úloh, kde je vyžadovaná dôvera zo strany príjemcu, čo môže byť napr. starý človek (ktorému robot podáva lieky, stará sa oň, či mu robí spoločníka). Cieľom humanoida tak nie je len vykonávať svoju prácu, ale aj dokonale imitovať človeka tak, aby sa naň čo najviac podobal a príjemca si k nemu mohol vybudovať hlbší vzťah a dôveru, tak ako u normálneho človeka. Humanoidní roboti môžu v podstate vykonávať akúkoľvek ľudskú prácu, teda prácu s nástrojmi. Vytvorenie softvéru pre riadenie je však náročné.

V neposlednom rade je tu etická otázka, do akej miery by sme mali vyvíjať humanoidných robotov. Ide hlavne o softvér, ktorý určuje "povahu"robota. Vo všeobecnosti však platí, že roboti s umelou inteligenciou by mohli byť využívané hlavne na nasadenie v podmienkach, ktoré sú pre človeka nebezpečné či náročné.

1.2 Lilli

Lilli (lIIIlI) je humanoidný robot navrhnutý Per R. Ø. Salkowitschom, projekt je open-source. Lilli má 25 stupňov voľnosti pohybu, teda 25 servomotorov. Je poskladaná z dreva a môže byť riadená ľubovoľným mikrokontrolérom podporujúcim I2C protokol alebo so schopnosťou riadiť 25 servomotorov, je možné využiť Arduino, Raspberry Pi či iný mikropočítač. Výška robota je 75 centimetrov.

Ako sme spomínali, Lilli je vyrobená z dreva, konkrétne z bukovej preglejky. Výhodou tejto konštrukcie je váha a cena. Pri poškodení dielu je veľmi jednoduché vytvoriť si jeho kópiu, prípadne modifikovať daný diel. Drevo má však aj svoje nevýhody; drevo nie je homogénne, teda v jeho vnútri sa môžu nachádzať nedokonalosti, ktoré môžu oslabovať daný diel a teda i celú konštrukciu. Taktiež, drevo je mäkšie ako väčšina kovov, napríklad hliník, teda má tendenciu viac sa ohýbať. V neposlednom rade, časté šraubovanie, teda rozoberanie a následné skladanie nie je najvhodnejšie.

Lilli taktiež nie je úplne prispôsobená na hardvér, ktorú ju obsluhuje či na "úložné"miesto komponentov. Ako príklad môžeme uviesť batérie umiestnené na hornej strane chodidiel. Väčším "úložným"miestom je hlava, toto miesto však nie je vhodné, pretože umiestnením ťažkých komponentov do hlavy posunieme aj ťažisko robota smerom hore, čo nie je, napríklad pre pohyb, ideálne.

Keďže Lilli je humanoidný robot, imituje ľudské telo, teda i možnosti pohybu. Azda najväčším "nedostatkom"je absencia prstov a brušných svalov. Rozloženie servomotorov je nasledovné:

- 6x v každej hornej končatine
- 5x v každej dolnej končatine
- 1x v torze
- 2x v krku

Horná končatina má tri servomotory v ramene, ktoré imitujú funkciu ľudského ramena a torzáciu predlaktia, jeden servomotor v lakti, ktorý slúži na zdvih predlaktia, a dva servomotory na torzáciu a otvorenie/zatvorenie manipulátora.

Na pohyb hlavy slúžia dva servomotory, jeden sa stará o otáčanie hlavy, druhý o jej záklon a predklon.

V hrudi sa nachádza jeden servomotor, ktorý zabezpečuje pohyb/stabilitu vrchnej časti tela.

V dolnej končatine sa nachádzajú tri motory v oblasti bedier. Tie zabezpečujú torzáciu, zdvih a rozkročenie dolnej končatiny. Následne, jeden servomotor sa nachádza v kolene a posledný sa nachádza v členku. Imitácia ľudských pohybov je teda na vysokej úrovni.

1.3 Pohyb u robotov

Existuje mnoho spôsobov, akými sa roboti pohybujú a taktiež pohyb robota je dôležitým aspektom pri jeho vývoji. Inšpiráciou je samozrejme aj pohyb živočíchov. Organizmy sa dokonale prispôsobili pre pohyb na prekonávanie každého terénu. Avšak napodobňovanie pohybu organizmov, resp. živočíchov, je veľmi náročné. Živé organizmy pracujú na bunkovej úrovni (napr. bunky svalov)pri ktorej dosahujú veľmi malý rozmer, váhu a mieru odolnosti. Navyše, tieto svalové systémy dosahujú lepšie vlastnosti ako človekom vytvorené systémy podobnej veľkosti. Väčšina robotov teda používa k pohybu kolesá alebo niekoľko kĺbových nôh (samozrejme využíva sa omnoho viac spôsobov lokomócie, ako je spomenuté). Vo všeobecnosti, pohyb zdieľa rovnaké otázky ohľadom stability, charakteristiku kontaktu s terénom a samotný typ terénu.

1.4 Chôdza u robotov

Hlavným problémom pri vývoji robotov, ktorý na pohyb (chôdzu) využívajú nohy je koordinácia pohybu ich nôh. V prípade robota s viacerými končatinami (nohami) ide o koordináciu nôh teda o spôsob chôdze. Počet možných spôsobov chôdze závisí od počtu nôh. Chôdza je sled udalostí zdvíhania a uvoľňovania pre jednotlivé nohy. Pre robota s k nohami je celkový počet odlišných sledov udalostí N pre kráčajúceho robota:

$$N = (2k - 1)! \text{ [25]}$$

Pre bipédneho robota s $k = 2$ nohami je počet odlišných sledov udalostí

$$N = (2k - 1)! = 3! = 6$$

Práve chôdza je náročná na celkovú stabilitu robota. Aj v stabilnej polohe musí robot vykonávať rôzne stabilizačné úkony pomocou motorov v oblasti bedier, kolien, členkov i vo vrchných častiach torza.

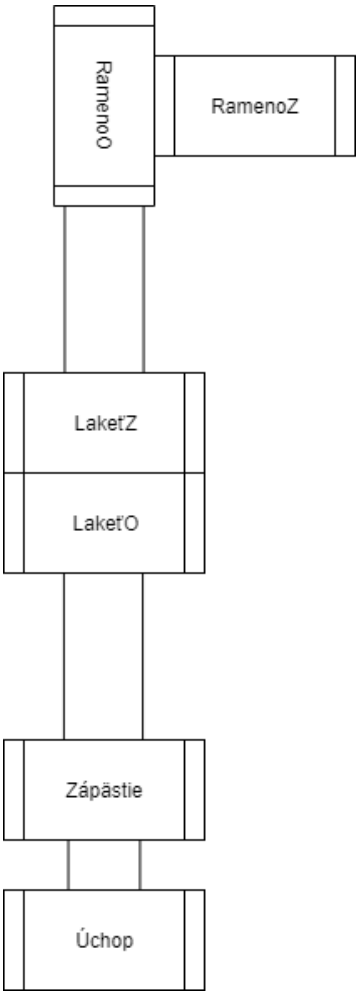
1.5 Kinematika horných končatín robota Lilli

V našom prípade sa jedná o kinematiku, ktorá pripomína rameno, keďže sa sústredíme na horné končatiny robota. V každej hornej končatine má Lilli 6 servomotorov, teda každá horná končatina má 6 stupňov voľnosti. Každý zo servomotorov má otočnosť približne 180°. Na schéme 1.1 vidíme rozloženie servomotorov. V ramene sa nachádzajú dva servomotory:

RamenoO a *RamenoZ*, pričom *RamenoO* otáča celou končatinou a *RamenoZ* zdvíha celú končatinu.

Podobne *LakťO* otáča celým predlaktím a *LakťZ* zdvíha celé predlaktie.

Zápästie následne otáča zápästím avšak nedvíha ho. *Úchop* otvára a zatvára "dlaň". Neprítomnosť tretieho stupňa voľnosti v zápästí sťažuje manipuláciu s objektami. Taktiež dĺžka medzi lakťom a ramenom je kratšia ako u človeka, čo spôsobuje kratší dosah oboch ramien do oboch strán. Citeľným problémom je manipulácia pred torzom, keďže dĺžka ramena je len o niečo väčšia ako je šírka torza robota.



Obr. 1.1: Pravé rameno

S týmito obmedzeniami je potrebné počítať už pri návrhu pohybu, ktorý sa počíta v danom softvéri.

1.6 Interakcia robota s okolím

Každý robot potrebuje k interakcii so svojím okolím senzory. Začína sa od najjednoduchších senzorov, ako sú napríklad dotykové senzory, či ultrazvukové alebo laserové senzory na zisťovanie vzdialenosti od objektu. Takéto senzory sa využívajú napríklad pri robotických vysávačoch, či jednoduchších robotických ramenách v priemyselnej výrobe. Mnoho problémov si však žiada dokonalejšie senzory na interakciu s okolím. Vo väčšine prípadov prichádza na rad istý druh spracovania obrazu.

1.7 Spracovanie obrazu

Spracovanie obrazu je už takmer nevyhnutnou súčasťou každého robota, ktorý rozpoznať objekty, prípadne aj ich farby, v priestore. Mnoho robotov vytvára hĺbkovú mapu priestoru, ktorý zaznamenáva prostredníctvom kamier na to určených. Robot následne vidí "3D"teda vie určiť vzdialenosť od jednotlivých objektov. Samozrejme, takáto technológia sa dá nahradiť aj senzormi vzdialenosti. Laserové technológie a senzory vedia taktiež rozpoznať tvary, softvér následne vie zreprodukovať a vizualizovať tieto dáta. Senzor na meranie vzdialenosti následne určí, v akej vzdialenosti sa daný objekt nachádza. Tým sa dá presne reprodukovať mapa priestoru v okolí robota. Laserové senzory ale pracujú len v jednej rovine. Laserové senzory, ktoré skenujú celý priestor sú výrazne drahšie. Nevýhodou laserových senzorov je ich technológia, teda ožarovanie priestoru laserom nie je vhodné pre človeka.

Vytváranie ofarbenej hĺbkovej mapy však má výhodu práve v rozpoznávaní farieb, čo môže byť veľkou výhodou pri vykonávaní rôznych činností, kde treba rozpoznať tvar, ale zároveň aj farbu objektu. Touto činnosťou je napríklad rozpoznávanie dielov v automobilovom priemysle, či rozpoznávanie ovocia a zisťovanie jeho stavu dozretia. Takáto hĺbková mapa napomáha aj pri interakcii s človekom.

1.8 Robotické videnie

Robotické videnie, zahŕňa využitie kamery (alebo viacerých kamier) a algoritmov na spracovanie dát zo vstupu kamery (kamier). Základným videním sú 2D kamery, ktoré vedia rozpoznať objekt (napr. automobilovú súčiastku pri využití robotov v automobilovom priemysle) a na základe toho vie robot, či má danú súčiastku zobrať, alebo nie (záleží od úlohy daného robota). Kamery však nemusia byť umiestnené priamo na robotovi (robotickom ramene).

V dnešnej dobe nie sú žiadnou výnimkou 3D kamery. Tie vytvárajú hĺbkovú mapu okolitého sveta. Tým pádom robot "vidí"podobne ako človek, teda vie povedať, v akej vzdialenosti sa od neho predmety nachádzajú. 3D kamery majú obrovské využitie v mnohých odvetviach, k príkladu napr. autonómne autá. Na robotické videnie v prípade Lilli sa využíva ZED Mini camera, ktorá zachytáva stereo video vo vysokom rozlíšení a napodobňuje tak spôsob, akým vnímame svet. Následným spracovaním vstupu z

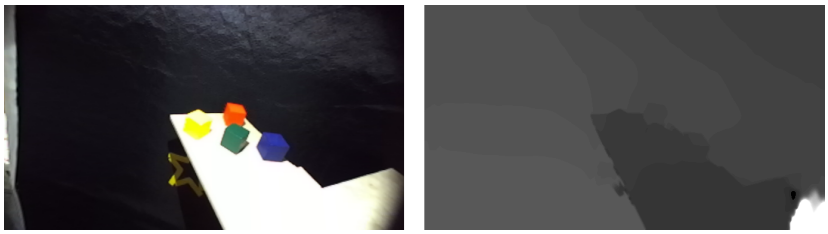
kamery sa vytvára ofarbená hĺbková mapa priestoru okolo Lilli.

1.9 Hĺbková mapa

Hĺbková mapa je výsledkom stereo videnia. Vzniká získaním informácie o hĺbke z dvoch (alebo viacerých) 2D obrázkov, podobne, ako vnímajú hĺbku priestoru aj ľudia.

Ako dáta na vstupe slúžia dva 2D obrazy. Tie sú zhotovené z dvoch rôznych pozícií v priestore. Kamery sú umiestnené na jednej priamke vo vzdialenosti v (báza alebo baseline). Keďže sú tieto kamery na rôznych pozíciách v priestore, aj zaznamenaný obraz bude odlišný, teda všetky objekty budú zaznamenané z odlišných uhlov (odlišnosť uhlov závisí od vzdialenosti v). Vzdialenosť objektov od kamier určuje aj ich pozíciu v zaznamenanom obraze. Keďže máme dva obrazy z rôznych uhlov, rozdiely medzi ich pozíciami v rámci obrazov budú odlišné, nenulové. Teda meraním relatívnych vzdialeností medzi objektami na obrazoch môžeme určiť vzdialenosť objektu.

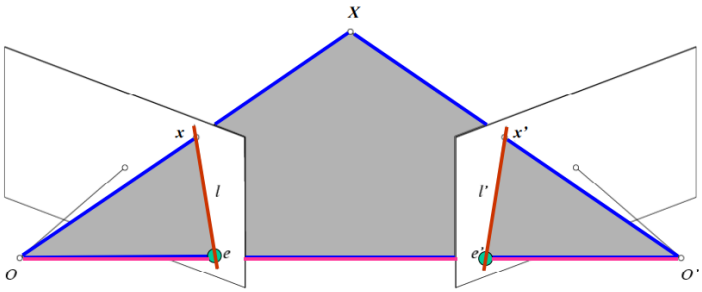
Toto tvrdenie však platí len pre objekty v konečnej vzdialenosti. Pokiaľ by sme uvažovali nekonečnú vzdialenosť, objekty by sa nám zobrazili na rovnakej pozícii na oboch 2D obrazoch.



Obr. 1.2: Príklad prostredia a jeho hĺbkovej mapy[14]

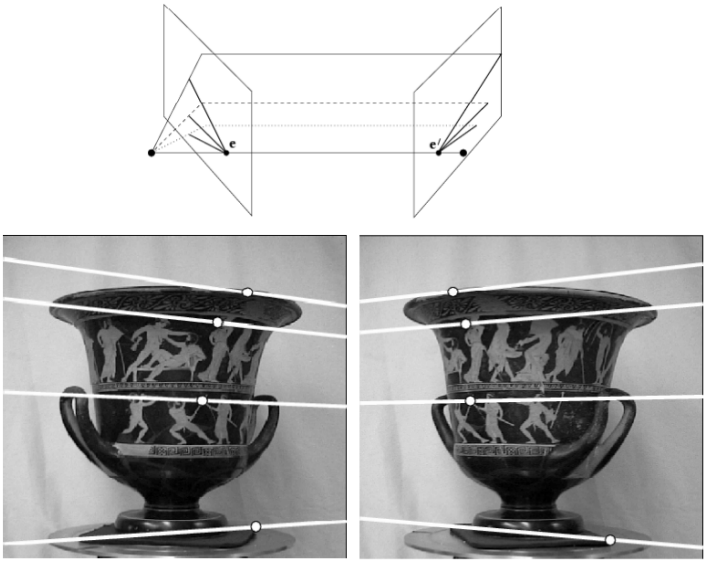
1.10 Epipolárna geometria

Epipolárna geometria popisuje geometrický vzťah medzi dvoma perspektívnymi pohľadmi na tú istú 3D scénu. Kľúčom je, že zodpovedajúce obrazové body musia ležať na konkrétnych obrazových priamkach. To znamená, že vzhľadom na bod na jednom obrázku je možné hľadať zodpovedajúci bod na druhom pozdĺž priamky a nie v oblasti 2D, čo predstavuje významné zníženie zložitosti.



Obr. 1.3: Epipolárna geometria [9]

Obrázok 1.3 znázorňuje epipolárnu geometriu. O a O' sú stredy projekcie kamier, ktoré ležia na základnej čiare (na 1.3 znázornená ružovou farbou). X je bod, ktorému chceme určiť hĺbku v obraze. Následne, rovina, ktorá prechádza stredom projekcie O, O' a objektom(bodom) X sa nazýva *epipolárna rovina* (na obrázku 1.3 znázornená modrou farbou).Bod X sa zobrazí na projekčnej rovine O a O' v bodoch x a x'. Nech sa bod X nachádza v akejkoľvek vzdialenosti od kamier, potom jeho obraz bude ležať na *epipolárnej čiare* l a l'. Tá vzniká prienikom roviny projekcie kamery a epipolárnej roviny. Ak projekčné roviny kamier nie sú paralelné, vzniká *epipól* e a e', priesečník projekčnej roviny a spojnice optických stredov kamier O a O'. [9] [22] [26]



Obr. 1.4: Využitie epipolárnej geometrie [9]

1.11 Mračno bodov

Mračno bodov je zoskupenie, rádovo, miliónov bodov v 3D priestore, teda každý bod má zadefinované svoje X, Y, Z súradnice. Pri mnohých 3D snímaniach sa zachytáva aj farba bodu, ktorá je uložená trojicou R, G, B v hodnotách 0 až 255. Tieto hodnoty sa získavajú z fotografií, nie zo samotného skenu. Veľkosť mračna závisí od snímaného priestoru či kvality snímky. Pri malom zábere okolia a nízkom rozlíšení snímača dosahuje mračno veľkosť rádovo niekoľko tisíc bodov. Pri profesionálnych 3D skenoch budov sa veľkosť mračna pohybuje v miliónoch až miliardách bodov. Spracovanie takéhoto množstva dát vyžaduje vysoký výpočtový výkon.

Po nasnímaní mračna bodov máme k dispozícii priestorový sken, z ktorého môžeme vytvoriť veľmi presný 3D model objektu.

1.11.1 Šum v mračne bodov

Ako takmer pri každom procese, aj pri zaznamenávaní mračna bodov vznikajú chyby. V našom prípade sa jedná o šum, teda o body, ktoré sú odľahlé od ostatných. Šum sa javí ako riedky zhluk bodov, ktorý je v kontraste s hustými zhlukmi skutočných objektov v mračne bodov. Tieto nepresnosti spôsobujú problémy s detekciou hrán či zakrivenia. Dôvodmi šumu v mračne bodov sú napríklad obmedzenia a nedokonalosti senzorov, zlé svetelné podmienky či vlastnosti odrazu svetla daného objektu (objekt môže byť príliš lesklý a teda javiť sa ako biela plocha, prípadne môže byť príliš tmavý a splývať s tmavým pozadím).

Pre ďalšie rozpoznávanie objektov v mračne bodov je teda nutné vyčistiť mračno od nežiadúcich bodov šumu. Na tento problém existuje niekoľko prístupov ako napríklad štatisticky založená metóda, metóda na základe porovnávania susedných bodov, prístupy filtrovania založené na projekcii či mnohé iné [17].

1.12 Robot Operating System

Robot Operating System (skrátene ROS) je framework alebo middleware pre programovanie robotov a písanie robotického softvéru. ROS bol vybudovaný za účelom vytvoriť robustný softvér na všeobecné použitie, ktorý rieši zložité problémy, ktoré sú pre človeka triviálne, avšak pre aplikáciu v robotike sú omnoho ťažšie; naprogramovanie

takéhoto softvéru je zložité a pre mnoho vývojárov až nereálne. ROS je open-source, čo pomáha jeho rastu.

I keď ROS nie je operačný systém, ponúka riešenia pre počítačový cluster ako napríklad hardvérovú abstrakciu, nízkoúrovňovú kontrolu zariadení, implementácia bežne používanej funkcionality, medzi procesorovú komunikáciu a správu balíkov[8]. ROS podporuje programovacie jazyky ako C++, Python či Lisp. Primárne bol určený na operačný systém Linux Ubuntu, zatiaľ čo ostatné systémy (Linux Fedroa, macOS, Microsoft Windows) boli označené ako "experimentálne". S príchodom ROS 2 sa však táto podpora rozšírila takmer na všetky operačné systémy.

1.13 Inverzná kinematika

Inverzná kinematika je opačným procesom ku doprednej kinematike. Pri probléme inverznej kinematiky poznáme pozíciu, do ktorej sa chceme manipulátorom dostať. Úlohou ostáva dopočítať uhly natočenia kĺbov (resp. motorov) tak, aby sme dostali manipulátor do novej, výslednej pozície.

Existuje niekoľko spôsobov, ako riešiť problém inverznej kinematiky:

- **Analytické riešenie**

  Analytické riešenie môže byť omnoho rýchlejšie ako numerické, môže poskytnúť viac, ako jedno riešenie, avšak vždy konečný počet riešení, ak existujú. Hlavným problémom analytického riešenia je problém, že pri zložitých robotických ramenách analytické riešenie neexistuje alebo je výpočtovo veľmi zložité a preto musíme používať numerickú aproximáciu. Taktiež, riešenia nemusia byť vždy najhladšie na vykonávanie a robotické rameno vykonáva neprirodzené pohyby.

- **Iteračné riešenie**

  Iteračné riešenia sú založené na postupnej aproximácii, teda na postupnom opakovaní výpočtu (funkcie) s cieľom postupne sa priblížiť k požadovanému výsledku. Iteračné riešenie môžeme ďalej rozdeliť na:

  - **Numerické riešenie**

    Pod numerickým riešením si môžeme predstaviť **Jacobiho maticu** (maticu parciálnych derivácií). Matica sa používa na zmenu uhlov kĺbu tak, aby sa príslušná končatina posunula do požadovanej polohy. V závislosti od poža-

dovanej účinnosti alebo presnosti môže byť Jacobiho matica niekoľkokrát prepočítaná, keď sa manipulátor pohybuje smerom k svojmu cieľu, alebo dokonca niekoľkokrát v priebehu pohybu smerom k svojmu cieľu[16].

– **Heuristické riešenie**

Heuristické algoritmy majú nízke výpočtové náklady (vo veľmi rýchlom čase dostávame výsledok) a zvyčajne podporujú obmedzenia kĺbov. Najpopulárnejšie heuristické algoritmy sú CDC a FABRIK[24].

1.14 Prehľad IK solverov

Keďže jedným z našich cieľov je uchopiť a premiestniť objekt, tento problém riešime pomocou inverznej kinematiky (1.13). Existujú rôzne, voľne dostupné riešenia v podobe aplikácií. Mnoho softvérov využíva na riešenie IK problémov ROS (1.12).

Poznámka: zoznam nezahŕňa všetky aplikácie na riešenie inverznej kinematiky. Vyhľadané aplikácie považujeme za vhodné na využitie v práci.

- **MoveIt**[3]

  MoveIt ponúka komplexné riešenie robotických systémov ako napríklad: plánovanie pohybu - vytvára trajektóriu pohybu cez prostredie, pričom sa vyhýba lokálnym minimám; manipulácia - umožňuje plánovať uchopenie objektu a jeho následnú manipuláciu; inverzná kinematika - výpočet inverznej kinematiky na umiestnenie ramena do správnej, požadovanej polohy; trojrozmerné vnímanie - umožňuje pripojiť niekoľko senzorov a pomocou nich rozpoznávať okolí; detekcia kolízií - vyhýbanie sa objektom v priestore.

  MoveIt ponúka podporu pre viac než 120 "komerčne"využívaných robotov, medzi ne patrí napríklad Robonaut, Valkyrie, Atlas, LBR KUKA, NAO či iCUB. MoveIt využíva ROS.

- **OpenRAVE**[4]

  OpenRave ponúka prostredie pre plánovanie, vyvíjanie a nasadzovanie algoritmov pre plánovanie pohybu. Najväčšie zameranie OpenRAVE je na simuláciu a analýzu pohybu (kinematiku) ramena/robota. Jeho výhodou je jednoduchá integrácia do už existujúcich robotických systémov.

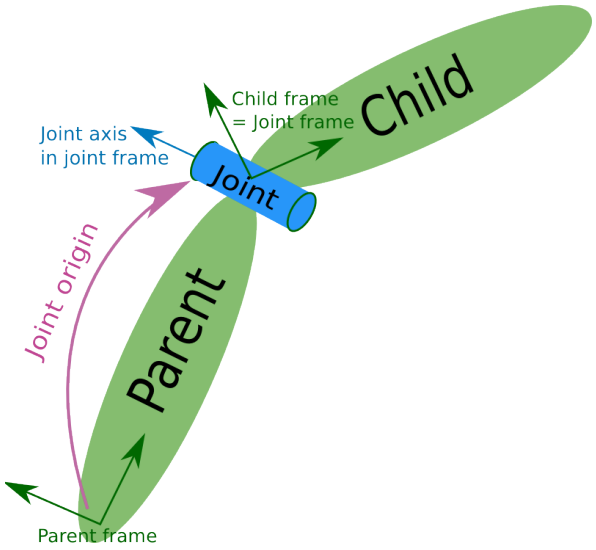
Azda najväčšou výhodou je technológia IKFast. Na rozdiel od väčšiny softvérov na riešenie inverznej kinematiky môže IKFast analyticky riešiť kinematické rovnice ľubovoľného komplexného kinematického reťazca a generovať jazykovo špecifické súbory (napr. v C ++) na neskoršie použitie. Konečným výsledkom sú mimoriadne stabilné riešenia, ktoré sú vypočítané v priebehu niekoľkých mikrosekúnd.

- **Orocos Kinematics and Dynamics**[6]
  Knižnica kinematiky a dynamiky (The Kinematics and Dynamics Library (KDL)) vyvíja aplikačne nezávislý framework pre modelovanie a výpočet kinematických reťazcov, ako napr. robotické ramená. Azda najväčšou výhodou je minimum požadovaných knižníc (na svoj chod potrebuje Eigen knižnicu). Pre svoj chod nepotrebuje ROS (je s ním však plne prepojiteľný). Implementačným programovacím jazykom je C++ a podporuje systémy Linux, macOS a Microsoft Windows.

- **CoppeliaSim**
  CoppeliaSim je robotický simulátor pomocou ktorého možno skúmať pohyby a správanie robota bez potreby fyzického stroja. V rámci diplomovej práce vytvoril Mgr. Gabriel Halasi[15] trojrozmerný model robota Lilli, ktorý následne načítal do CoppeliaSim. Ten následne môžeme využiť na výpočet inverznej kinematiky ramena pre uchopenie objektu (a taktiež na počiatočné pokusy pre pohyb robota).

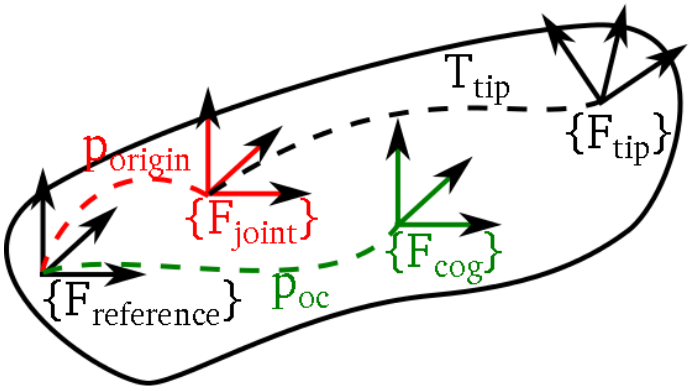
1.15 Kinematic tree a URDF

K dispozícii máme URDF pre Lilli[15]. v URDF je rameno definované ako postupnosť linkov a jointov, ktoré sa striedajú. V URDF nás zaujímajú len pointy, pretože všetky translácie a rotácie uvedené v linkoch sa týkajú len lokálnych vlastností linkov: pozícia ťažiska voči súradnicovému systému linku, pozícia zobrazeného STL súboru (vizualizácie) voči súradnicovému systému linku a pozícia STL súboru použitého na výpočet kolízií voči súradnicovému systému linku. Naopak v popise Joint-u sa udáva zmena súradnicového systému medzi predchádzajúcim linkom a nasledujúcim linkom, ktoré joint spája, čiže de-facto vektor predchádzajúceho linku a potrebná rotácia medzi súradnicovými systémami predchádzajúceho a nasledujúceho linku.



Obr. 1.5: Zobrazenie jointu v URDF [7]

V Kinematic Tree KDL 1.1 je rameno definované ako tzv. chain, ktorý pozostáva zo segmentov. Každý segment má svoj vektor (čiže posun medzi súradnicovými systémami zodpovedajúceho linku a nasledujúceho linku) a rotáciu, ktorá je medzi príslušným linkom a nasledujúcim linkom (čiže pod akým uhlom je napojený nasledujúci link na konci príslušného linku).



Obr. 1.6: Zobrazenie KDL Tree [5]

Preto pri konštrukcii segmentov kinematickej reťaze pre KDL budeme postupovať nasledovne: každý segment je určený hodnotami translácie a rotácie, ktoré sú uvedene v jointe, ktorý v URDF nasleduje za príslušným linkom, ktorému segment zodpovedá.

1.16 Point Cloud Library - PCL

Po vytvorení mračna bodov ho potrebujeme spracovať. Jednou z možností je Point Cloud Library. PCL je open-source knižnica algoritmov na spracovanie 2D obrazov a 3D mračien bodov vyvíjaná mnohými nadšencami, spoločnosťami i univerzitami po celom svete. Je vyvíjaná v rámci BSD licencie, teda pre výskumné či komerčné účely je zdarma. Vývojovým programovacím jazykom je C++.

PCL je členená do niekoľkých modulárnych knižníc, ktoré je možné medzi sebou kombinovať a väčšinou poskytujú postupnosť krokov pri detekcii a spracovaní mračna bodov. Obrovskou výhodou PCL je nezávislosť od zariadenia a podpora mnohých 3D formátov ako *.xyz*, *.pcl*, *.ply*, či *.pcd*.

1.17 OpenCV

OpenCV, Open Source Computer Vision Library, je knižnica algoritmov zameraná hlavne na počítačové videnie a spracovanie obrazu v reálnom čase[23]. OpenCV bolo vyvíjané od roku 1998 pôvodne spoločnosťou Intel. Verejnosti však bolo predstavené až v roku 2000. V súčasnosti sa vyvíja pod licenciou BSD a podporuje takmer všetky operačné systémy. Primárnym programovacím jazykom je C/C++, má však podporu aj pre jazyky Python a Java.

Podobne ako Point Cloud Library, OpenCV obsahuje niekoľko "modulov"pre spracovanie 2D a 3D obrazu a taktiež podporuje akceleráciu grafickým procesorom. Asi najznámejšie sú algoritmy na segmentáciu obrazu, vytváranie hĺbkových máp či rozpoznávanie tvárí v obraze.

1.18 OpenCV a Point Cloud Library

Na prvotné rozpoznávanie objektov v obraze sme používali OpenCV. Počiatočný algoritmus bol veľmi jednoduchý:

- Z vyhotovenej fotografie sme spravili čiernobiely obrázok podľa konštánt, ktoré od daného konštantného bodu (jasu obrazu na danom pixely) rozhodnú, či pixel bude čiernej alebo bielej farby.

- Pomocou vektorov sme vytvorili kontúry objektu, teda dostali sme biele 2D útvary na čiernom pozadí. Kontúry boli presné a jasne definovali objekty v scéne.

Tretím krokom by bolo spustenie algoritmu na rozpoznávanie útvarov v scéne. Ešte pred týmto krokom sme však zaznamenali niekoľko problémov. Hlavným problém nastával pri zlých svetelných podmienkach; bledé objekty splývali s bledým podkladom, na ktorom boli umiestnené. Tento problém sme sa snažili odstrániť nastavením jasu a kontrastu kamery, avšak ani tieto úpravy neboli dostatočné a svetlé objekty sa častokrát strácali zo scény.

Nie nutným problémom, ale určite možným nedostatkom, bolo rozpoznávanie objektov v scéne; išlo iba o rozpoznávanie 2D útvarov, ktoré, ak bol objekt, napr. kocka, natočený, algoritmus by mohol útvar rozpoznať ako kosoštvorec. Poslednou nevýhodou bolo získavanie vzdialenosti objektu, ktoré je pri využití mračna bodov v spojení s Point Cloud Library omnoho jednoduhšie, ako pri využití OpenCV.

1.19 NVIDIA Jetson TX2

Na spracovanie obrazu využívame NVIDIA Jetson TX2[18] - (malý) superpočítač na prácu s umelou inteligenciou, ktorý je schopný spracovať veľké množstvo dát v krátkom čase. Jeho základným kameňom je grafický procesor Pascal s 256 jadrami, ktoré sa využíva v grafických kartách podporujúcich virtuálnu realitu, prácu s umelou inteligenciou či s najnáročnejším grafickými programami. Poháňaný je štvorjadrovým procesorom s ARM architektúrou. Výhodou je taktiež spotreba. Tá sa pohybuje od 7W do 15W (pri plnej záťaži), na jeho chod teda postačí aj batéria. Operačným systémom je Linux Ubuntu 16.04.

NVIDIA Jetson TX2 taktiež podporuje technológiu NVIDIA Cuda, ktorá umožňuje na grafickom procesore spúšťať programy vytvorené v programovacích jazykoch C/C++ a Fortran. Táto grafická akcelerácia je veľmi vhodná pri práci s point cloudom, ktorý musí spracovať rádovo milióny bodov v 3D priestore v priebehu niekoľkých sekúnd. Keďže v práci využívame Point Cloud Library a programovací jazyk C++, môžeme plne využiť grafickú akceleráciu.

1.20 ZED mini

Očami Lilli je kamera ZED mini. Kamera v sebe zahŕňa niekoľko funkcionalít ako napríklad vytvorenie hĺbkovej mapy, zaznamenanie mračna bodov či nástroje na rozšírenú reaitu. Kamera sa skladá z dvoch objektívov, ktoré pracujú na

1.21 Random sample consensus - RANSAC algoritmus

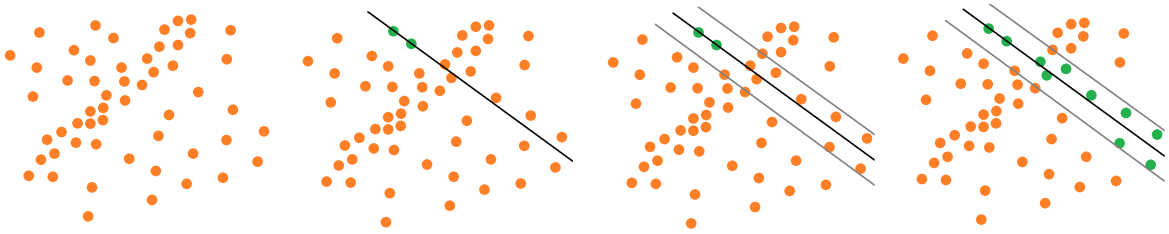
Algoritmus RANSAC je jeden z mnohých algoritmov na detekciu bodov, ktoré ležia mimo hlavného zhluku bodov (ang. outliers), či už v 2D alebo 3D scéne, ktoré chceme skúmať. Príkladom môže byť detekcia roviny, kde body roviny sú žiadúce body, patriace öbjektu", ktorý chceme získať (ang. inliers) a ostatné body sú nežiadúce (outliers), keďže nepatria nami skúmanému objektu, teda rovine. Algoritmus bol prvýkrát publikovaný v roku 1981 vedcami Fischlerom and Bollesom.

RANSAC je založený na metóde pokus-omyl, teda je založený na riešení problému pomocou náhodného testovania všetkých potencionálnych riešení. V praxi to znamená, že aj pri 50% bodov ležiacich mimo naše kritériá je algoritmus stále schopný nájsť body, ktoré patria nami hľadanému objektu. Hlavnou myšlienkou je nájsť čo najlepšie rozdelenie bodov patriacich objektu a bodov, ktoré doň nepatria. Algoritmus na začiatku ühádne", resp. zvolí si, niekoľko bodov a následne preskúma všetky ostatné body, ktoré ešte neboli použité. Preskúmaním ich ohodnotí, čím dostávame označenie, či bod patrí alebo nepatrí hľadanému objektu. Tento postup sa opakuje niekoľkokrát, všeobecne platí, čím viac iterácií vykonáme, tým väčšiu presnosť dosiahneme. Pri tomto algoritme nastavujeme niekoľko premenných ako napríklad počet iterácií či maximálnu vzdialenosť bodov medzi sebou aby boli body považované za susedov, teda patriace k objektu. Kroky algoritmu teda sú:

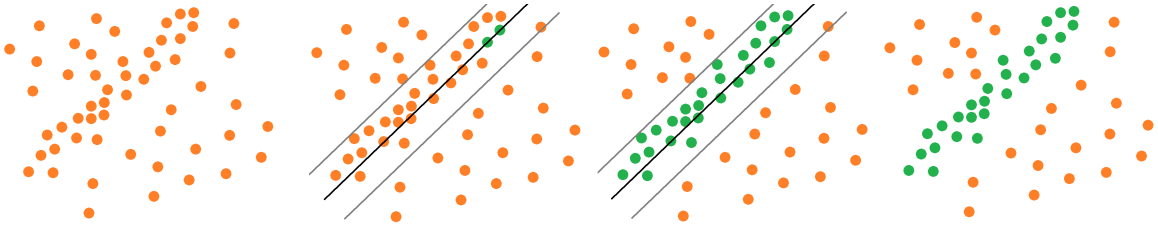
1. **Zvolenie si náhodných bodov** ktoré sú označené ako body patriace objektu. V tomto kroku sa vytvorí spojnica týchto bodov, teda všetky zvolené body budú ležať na priamke
2. **Výpočet** bodov patriacich objektu, teda preskúmame okolie od priamky vo vzdialenosti δ. Všetky body, ktoré sa nachádzajú v tomto pásme, sú body, ktoré

patria objektu

3. **Spočítanie** bodov patriacich objektu, teda spočítanie všetkých bodov, ktoré ležia v našom prehľadávanom pásme



Obr. 1.7: Zobrazenie fungovania algoritmu RANSAC



Obr. 1.8: Zobrazenie fungovania algoritmu RANSAC - v tomto kroku však algoritmus vybral vhodnejšie body a našiel objekt

Algoritmus spúšťame v nami zvolenom počte iterácií a výsledkom sú dáta s najvyšším počtom bodov (krok 3). Dôležitým krokom algoritmu je správna voľba počtu iterácii, nízky počet iterácií nemusí detegovať objekt správne resp. algoritmus vytvorí veľa šumu, keďže označí mnoho bodov ako patriace objektu. Veľký počet iterácií môže spomaliť výpočet a taktiež viesť k nepresnostiam. Ideálny počet iterácii vypočítame pomocou nasledovnej rovnice:

$$1 - p = (1 - w^n)^k$$

kde p je požadovaná pravdepodobnosť, podľa ktorej RANSAC poskytne uspokojivý výsledok rozpoznávania, k je počet iterácií, w je pravdepodobnosť výberu bodu patriacemu objektu pri každom výbere bodu; w = počet patriacich bodov objektu/počet všetkých bodov, hodnotu w však nevieme vypočítať dopredu, vieme však určiť hrubý odhad. Teda n je počet bodov, podľa ktorého vieme určiť objekt, w^n je pravdepodobnosť, že všetky body patria objektu a $1 - w^n$ je pravdepodobnosť, že aspoň jeden z n bodov je bod ležiaci mimo objektu. Po upravení vzorcu vyššie dostávame:

$$k = \frac{log(1-p)}{log(1-w^n)}$$

Pri zadávaní n platí - čím menšie, tým presnejší výsledok, avšak na úkor časového spracovania. Ani táto hodnota počtu opakovania iterácií však nie je najvhodnejšia, preto by sme ju mali využívať ako horný limit iterácií.

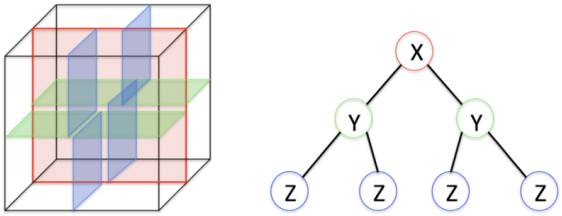
Výhody algoritmu sú robustné spracovanie bodov ležiacich mimo objektu, vysoká miera parametrizácie či jednoduchá implementácia. Nevýhodami je rýchlosť výpočtu, ktorý rastie spolu s veľkosťou dát a veľkosťou parametra n. Algoritmus taktiež nie je vhodný na rozpoznávanie viacerých objektov naraz. Podobným algoritmom, avšak na oválové útvary, je Houghova transformácia.

1.22 Kd-tree

Kd-tree, alebo k-dimensional tree (k-rozmerný strom), je štruktúra na usporiadanie bodov podľa rozdelenia k rozmerného priestoru (najčastejšie Euklidovského priestoru). k-rozmerný strom je binárny strom kde každý vrchol predstavuje rozdelenie bodov podľa jeden z osí priestoru. V našom prípade sa $k = 3$ keďže pracujeme s trojrozmerným priestorom mračna bodov. k-rozmerný strom vytvoríme nasledovným spôsobom:

1. Nájdeme medián x-ových hodnôt bodov. Podľa tohoto čísla rozdelíme priestor na dve časti.
2. V každej časti nájdeme medián y-ových hodnôt bodov. Každý priestor rozdelíme podľa tejto hodnoty na ďalšie priestory.
3. Tieto priestory rozdelíme pomocou mediánu z-ových hodnôt bodov. Dostaneme tak nové priestory.

Medián predstavuje vrchol stromu. Algoritmus pokračuje dovtedy, kým v každom priestore nezostane iba jeden bod. Dostávame teda hotovú štruktúru k-rozmerného stromu.



Obr. 1.9: Zobrazenie rozdelenia priestoru - vizualizácia Kd-tree [12]

Kapitola 2

Návrh a implementácia riešenia

2.1 Pohyb (chôdza) robota

Keďže pracujeme s humanoidným robotom, chceme, aby jeho pohyby boli čo najpodobnejšie tým ľudským. To platí aj pre chôdzu (1.4). Prvým problémom pri tejto úlohe je stabilita robota. Ľudia si pri chôdzi neuvedomujú, koľko "balansovacích procesov"vykonávajú; chôdza je pre nás prirodzená. Pri každom kroku však treba korigovať telo, aby sa nepreválilo dopredu, dozadu či do strán.

Základným cieľom je udržať torzo v upriamenej polohe (robot stojí na mieste). I tento úkon je náročný. Naivným prístupom je využitie protipohybu, teda, ak sa robot nakloní o 10° dopredu, v príslušnom kĺbe (alebo viacerých kĺboch) vykonáme pohyb v protismere, aby sme robota dostali no pôvodnej pozície (teda náklon 0°). Tu však narážame na niekoľko problémov.

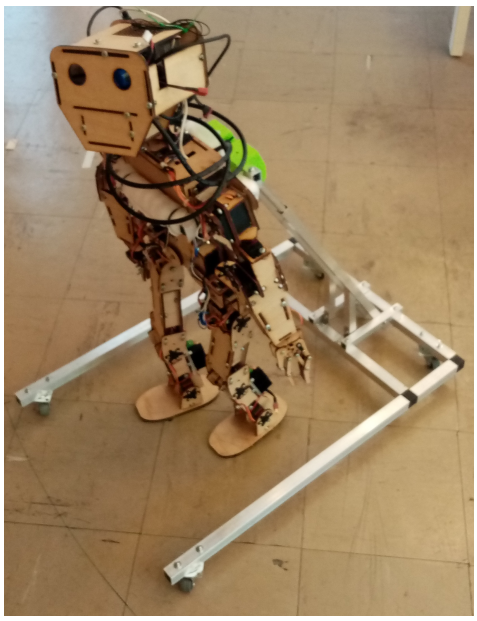
Prvým je **zotrvačnosť** torza. Pokiaľ urobíme prudký pohyb v protismere pôvodného pohybu, rýchlo sa dostaneme do pôvodnej polohy, avšak nezostaneme v nej a pokračujeme ďalej, už v novom pohybu, ktorý nás dostáva preč, od požadovanej polohy. Opäť sa dostaneme do situácie, kedy sa dostaneme do pôvodnej polohy a prejdeme do nechceného pohybu. Týmto spôsobom sa robot rozkmitá a nasleduje jeho pád. Druhým problémom je **konštrukcia a ťažisko** robota. Ako sme popísali v sekcii (1.2), ťažisko robota sa nachádza vyššie ako u človeka, teda nachádza sa približne v oblasti hrude pričom ťažisko človeka sa nachádza v oblasti pásu. To spôsobuje väčšiu náchylnosť k prevráteniu sa.

Ďalším problémom je **konštrukcia** robota (1.2). Ohybnosť dreva sťažuje balansova-

nie i samotnú chôdzu. Nevieme totiž stopercentne predpovedať pozíciu robota, jemný ohyb dreva spôsobuje nechcený pohyb a náklony.

Problémom sú aj samotné **servomotory**. Ich sila nebola dostatočná na udržanie robota (hlavne motory v dolných končatinách). Čiastočným riešením problému bola výmena štyroch servomotorov v kolenách a členkoch robota.

Kvôli vyššie spomenutým problémom sme sa rozhodli vytvoriť chodítko na podporu chôdze a balansovania.



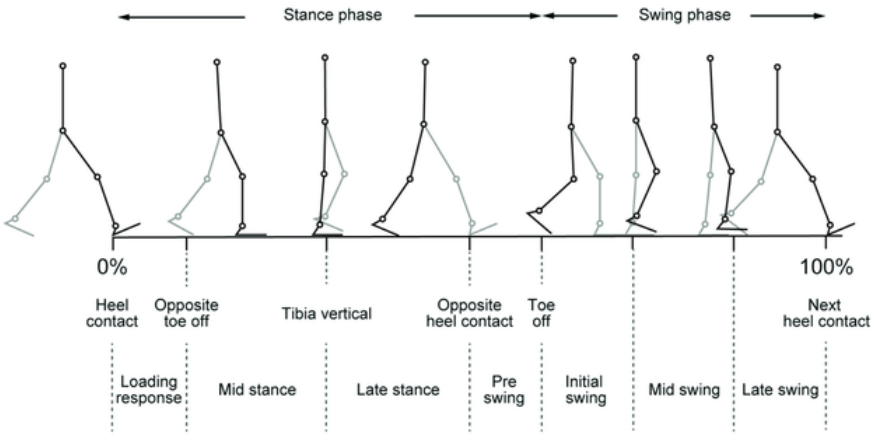
Obr. 2.1: Chodítko pre Lilli [1]

2.1.1 Návrh algoritmu chôdze

Algoritmus vychádza z imitácie ľudskej chôdze avšak musíme počítať s niektorými limitáciami. Východisková pozícia je stánie. Algoritmus má nasledujúce fázy:

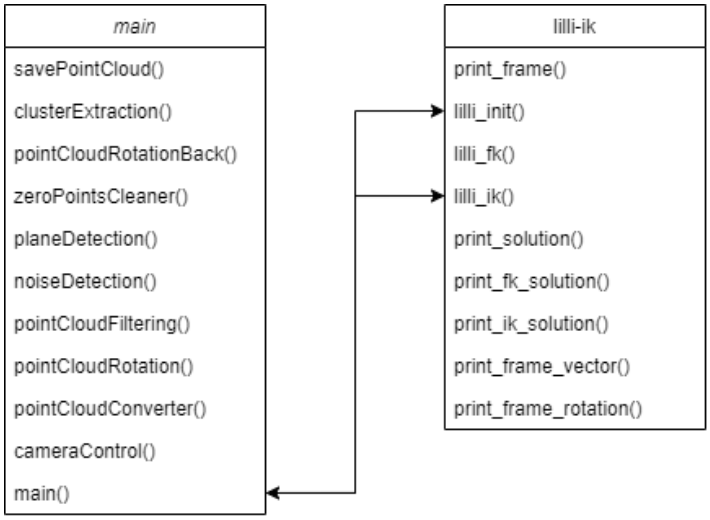
1. Prenesenie váhy na ľavú nohu
2. Vystieranie pravej nohy v bedre a súčasne ohyb v kolene
3. Vyrovnanie sa v bedrách
4. Ohýbanie ľavej nohy v bedre a kolenách tak, aby sme opäť boli na oboch nohách
5. Prenesenie váhy na pravú nohu a postupne vystieranie pravej nohy v bedre a kolene
6. Prisúvanie ľavej nohy vpred

7. Prenesenie váhy na ľavú nohu a jej vystretie
8. Prisunutie pravej nohy a jej presun vpred



Obr. 2.2: Návrh algoritmu chôdze [20]

2.2 Triedny diagram



Obr. 2.3: Triedny diagram riešenia

2.3 O implementácii

Ako implementačný programovací jazyk sme zvolili C++; keďže pracujeme s veľkým množstvom dát, potrebujeme efektívny programovací jazyk, ktorý dokáže v krátkom čase spracovať tieto dáta. Najväčšiu časovú záťaž pociťujeme v úvode spracovania

mračna bodov, keďže vtedy je mračno najväčšie a neopracované. Kvôli rýchlosti spracovania a dosahu rúk Lilli je mračno neskôr orezané približne o 95% bodov v mračne. Detegovanie objektov potom prebieha rýchlo a pomerne presne. Taktiež, používaná knižnica má najväčšiu podporu práve v C++. V práci využívame knižnicu Point Cloud Library; práve vďaka nej nie je implementácia dlhá, keďže pracujeme s dobre vyvinutou knižnicou.

V časti Inicializácia kamery a získanie mračna bodov (2.4) vysvetlíme metódu, pomocou ktorej získame z kamery ZED mini mračno bodov. V časti Spracovanie mračna bodov (2.5) vysvetlíme ako spracujeme mračno bodov. Stretávame sa s niekoľkými prekážkami, ktoré musíme vyriešiť, aby sme mohli správne detegovať objekt. Prvou je otočenie mračna bodov (2.5.1), tento krok je potrebný na vyrovnanie mračna do prirodzenej polohy. Keďže Lilli má obmedzený priestor, v ktorom dočiahne manipulovať s objektami, nepotrebujeme celé mračno bodov, preto ho orežeme (2.5.2). Tým ušetríme čas spracovania mračna v ďalších krokoch a spresníme detekciu. Keďže pri snímaní mračna vzniká šum (1.11.1), musíme ho odstrániť (2.5.3). Aby sme boli schopný detegovať objekty, potrebujeme sa najprv zbaviť roviny (2.5.4). Na to využívame algoritmus RANSAC (1.21). V sekcii Detegovanie objektov (2.6) popíšeme metódu na detegovanie objektov (2.6.1, 1.9). V sekcii Pomocné funkcie (2.7) popíšeme pomocné metódy, ktoré zabezpečujú konverziu dátových typov mračna bodov, odstránenie prebytočných nulových bodov a uloženie mračna bodov pre vizualizáciu.

Ďalej sa budeme venovať inverznej kinematike (2.8). Na začiatok si musíme inicializovať (2.8.2) kinematickú reťaz vytvorenú pomocou URDF (1.15). Následne môžeme spustiť inverznú kinematiku (2.8.4), ktorá nám vypočíta natočenie servomotorov pre dosiahnutie objektu. Taktiež máme k dispozícii doprednú kinematiku (2.8.3).

2.4 Inicializácia kamery a získanie mračna bodov

2.4.1 void cameraControl()

Funkcia **void cameraControl()** inicializuje kameru ZED mini. Po inicializácii nastáva konfigurácia kamery. Ako prvé sa nastavuje rozlíšenie. Maximálne rozlíšenie ZED mini je 2.2K teda 4416x1242 px pri 15 snímkach za sekundu. Pri nižších rozlíšeniach samozrejme počet snímkov za sekundu narastá, avšak pre našu aplikáciu je vhodné čo

najvyššie rozlíšenie, keďže snímame statické objekty. Vyššie rozlíšenie prináša väčšiu presnosť mračna bodov, čo je v našom prípade najdôležitejšie. Kľúčovým nastavením kamery je nastavenie hĺbkového rozsahu, teda minimálnej (a maximálnej) vzdialenosti, pri ktorej je možné odhadnúť hĺbku objektu. Predvoleným minimálnym nastavením kamery je hodnota 0.4 metra čo je v našom prípade príliš vysoká hodnota. Pre našu potrebu nastavujeme hodnotu na 0.15 metra (teda na najnižšiu možnú hodnotu). Nízku hodnotu nastavujeme z dôvodu, že objekty sa nachádzajú 20 až 30 centimetrov pred kamerou. Táto vzdialenosť je daná veľkosťou rozsahu rúk Lilli, ktorá sa pohybuje na úrovni ~ 25 centimetrov.

Následne nastavujeme úroveň snímania hĺbky obrazu, ktorú nastavujeme na $ULTRA$; opäť dosahujeme najväčšiu presnosť na úkor snímkov za sekundu, čo nám nevadí. Úroveň snímania nastavujeme na $FILL$ čo nám zabezpečí presnejší obraz.

Posledným krokom je získanie hĺbkovej mapy, ktorá je uložená ako mračno bodov vo formáte XYZ.

2.5 Spracovanie mračna bodov

2.5.1 void pointCloudRotation()

void pointCloudRotation() zabezpečuje rotáciu mračna bodov podľa osi x. Mračno rotuje o θ radiánov, v našom prípade 0.5 radiánov (30°), čo je uhol, pod ktorým je natočená kamera (resp. hlava Lilli) vzhľadom na rovinu scény. Rotácia je potrebná z dôvodu ľahšieho segmentovania mračna bodov a následného získavania koordinátov objektov v trojrozmernom priestore. Pod rotáciou rozumieme rotáciu matice. Rotáciou chceme docieliť, aby sa body x, y, z zobrazili do bodov bod daným uhlom. To však nevieme docieliť maticou o veľkosti $3x3$, musíme preto pracovať s maticou o veľkosti $4x4$.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2.5.2 void pointCloudFiltering()

void pointCloudFiltering() vyvára filtrovaný objekt, teda filtrované mračno bodov. V jednoduchosti môžeme povedať, že dôjde k odrezaniu časti mračna. Tento krok sa javí ako hrubý avšak má svoje opodstatnenie i výhody. Prvým filtrovaním prejdú body na osi x. Odstránia sa krajné body čím odstránime takmer polovicu celkovej šírky mračna. Odstránené časti sú pre nás nepodstatné z dôvodu, že k nim Lilli nedočiahne, teda objekty sú mimo dosahu robotických ramien.

Následne filtrujeme mračno bodov podľa osi z a taktiež odstránime časti, kam sa nevieme dostať robotickým ramenom, teda časti, ktoré sú vo vzdialených častiach scény.

Ako posledná prichádza filtrácia podľa osi y. Odstránením týchto "nepotrebných"bodov urýchľujeme ďalšie funkcie, ktoré ďalej pracujú s mračnom bodov.

2.5.3 void noiseDetection()

Ďalším krokom je funkcia **void noiseDetection()**, ktorá slúži na filtráciu šumu(1.11.1) v mračne bodov. Po filtrácii ostáva očistené mračno bodov, ktoré nám umožňuje presnejšiu prácu v nasledujúcich krokoch.

Na začiatku funkcie si vytvoríme filter *StatisticalOutlierRemoval* a nastavíme počet susedných bodov, ktoré sa budú analyzovať pre každý bod, a multiplikátor smerodajnej odchýlky na 1. Inak povedané, pre každý bod sa skúma 50 bodov v jeho okolí, pokiaľ je smerodajná odchýlka daného bodu väčšia ako 1, bod je označený ako šum. Teda zvolený bod leží ďaleko od ostatných bodov a nepatrí k žiadnemu väčšiemu zhluku bodov v mračne.

Po preskúmaní všetkých bodov nám ostávajú len body ležiace vo väčších zhlukoch bodov v mračne, teda očistené mračno bodov.

2.5.4 void planeDetection()

Úlohou tejto funkcie je detegovať a odstrániť najväčšiu rovinu v scéne, v našom prípade ide o stôl, na ktorom sú položené objekty určené na detekciu.

Prvým krokom je inicializácia segmentovaného objektu
$pcl :: SACSegmentation < pcl :: PointXYZ >$. Následne definujeme typ jeho modelu a metódy. Typom modelu je $pcl :: SACMODEL\_PLANE$, teda model roviny (iné

modely sú napríklad priamky, kruhy, gule, valce, kužele, ...). Ako typ metódy sme si zvolil *pcl* :: *SAC_RANSAC*, inak povedané, rovinu budeme detegovať algoritmom *RANSAC* (1.21). Posledným krokom je určenie prahu, resp. veľkosti okolia, v ktorom sa bude detegovať zhluk bodov roviny. Výsledkom je mračno bodov roviny spolu s parametrami roviny v tvare $\mathbf{ax + by + cz + d = 0}$.

Pre získanie mračna bodov bez roviny nastavíme body, ktoré patria roviny, v pôvodnom mračne na 0, teda na súradnice 0,0,0. Týmto krokom však vygenerujeme množstvo "prázdnych"bodov, ktoré pri detegovaní objektov spomaľujú tento proces a vedú k nepresnosti. Tieto body treba z mračna odstrániť.

2.6 Detegovanie objektov

2.6.1 void clusterExtraction()[2]

Posledným krokom je samotná detekcia objektov v scéne. Priebeh algoritmu je nasledovný:

1. Vytvoríme k-rozmerný strom (1.9) pre vstupné mračno bodov $\mathbf{P}$
2. Inicializujeme prázdnu dátovú štruktúru *list* pre zhluky $\mathbf{C}$ a front bodov, ktoré treba skontrolovať $\mathbf{Q}$
3. Následne pre každý bod $\mathbf{p_i} \in \mathbf{P}$ postupujeme nasledovne:
   - $\mathbf{p_i}$ pridáme do fronty $\mathbf{Q}$
   - Pre každý bod $\mathbf{p_i} \in \mathbf{Q}$ vykonáme:
     - Hľadáme množinu $\mathbf{P_i^k}$ bodov susedov bodu $\mathbf{p_i}$ v okolí v tvare gule s polomerom $\mathbf{r < d_{th}}$
     - Pre každého suseda $\mathbf{p_i^k} \in \mathbf{P_i^k}$ skontrolujeme, či daný bod bol skontrolovaný, ak nie, pridáme ho do fronty $\mathbf{Q}$
   - Keď sme spracovali všetky body v $\mathbf{Q}$, pridáme $\mathbf{Q}$ do listu $\mathbf{C}$ a ostatok $\mathbf{Q}$ priradíme do prázdneho listu
4. Algoritmus sa zastaví, keď sme spracovali všetky body $\mathbf{p_i} \in \mathbf{P}$ a sú súčasťou $\mathbf{C}$

Výsledkom sú teda jednotlivé objekty. Tie si následne ukladáme do vektora. V ďalšom vektore si ukladáme stred detegovaného objektu. Tieto súradnice následne posie-

lame do časti riešenia inverznej kinematiky, ktorá vypočíta pohyb ramena smerom k objektu.

2.7 Pomocné funkcie

2.7.1 void pointCloudConverter()

void pointCloudConverter() zabezpečuje prevod mračna bodov z formátu *sl::Mat slPointCloud* do formátu *pcl::PointCloud<pcl::PointXYZ>*. ZED mini využíva prvý spomínaný formát a Point Cloud Library využíva druhý spomínaný formát.

2.7.2 void zeroPointsCleaner()

Funkcia slúži na odstránenie "nulových"bodov v mračne, teda bodov, ktorých súradnice x, y a z sú rovné nule (0,0,0). Takéto body vznikajú ako reprezentácia bodov, ktoré sú príliš blízko, či príliš ďaleko od kamery. V našom prípade taktiež odstraňujú umelo vzniknuté nulové body 2.5.4.

2.7.3 void savePointCloud()

Funkcia **void savePointCloud()** nám umožňuje export mračna bodov v akomkoľvek bode spracovania mračna. Exportované dáta je následne možné vizualizovať v mnohých programoch na zobrazovanie 3D scén.

2.8 Inverzná kinematika

2.8.1 void print_frame()

Funkcia **print_frame()** slúži na vypísanie rámca (Frame) v ktorom sa uschováva matica - predstavuje transformáciu rámcov v 3D priestore (rotácia + translácia).

2.8.2 void lilli_init()

Keďže Orocos nám neumožňuje načítať URDF formát (Unified Robot Description Format), musíme manuálne na začiatku zadefinovať kinematickú reťaz, v našom prípade

robotické rameno s 5 stupňami voľnosti. Následne si vytvoríme Eigen maticu - do nej pridávame premenné L, ktoré nám hovoria, aký atribút uprednostniť pri výpočte inverznej kinematiky.

2.8.3 init lilli_fk()

Táto funkcia slúži na využitie rekurzívneho algoritmu doprednej kinematiky a na výpočet transformácie pozícií zo všeobecného priestoru do karteziánskeho priestoru všeobecného kinematického reťazca.

2.8.4 int lilli_ik()

Funkcia **lilli_ik()** je pre našu aplikáciu najpotrebnejšia. Jej úlohou je vypočítať inverznú kinematiku ramena, podľa vypočítaných údajov ho následne navigujeme k objektu. Funkcia na výpočet inverznej kinematiky využíva Levenberg–Marquardt algoritmus. Robustnosť a rýchlosť algoritmu je dosiahnutá niekoľkými spôsobmi ako napríklad samotné využitie LMA algoritmu, ktorý automaticky zefektívňuje výpočet, vnútorným počítaním doprednej kinematiky, ktorá pomáha k robustnosti, či vďaka skutočnosti, že iba konštruktor volá dynamickú alokáciu pamäte.

Funkcie void print_solution(), print_fk_solution(), void print_ik_solution(), void print_frame_vector(), void print_frame_rotation() slúžia na kontrolné výpisy riešenie, doprednej kinematiky, inverznej kinematiky, vectoru a translácie (menované v poradí).

Kapitola 3

Testovanie a výsledky testovania

3.1 Detegovanie objektov

Našou úlohou bolo detegovanie objektov reprezentovanými farebnými drevenými kockami o rozmeroch $\sim 2x2x2$ centimetra. Farby kociek boli červená, zelená, modrá a žltá. Kocky boli umiestnené pred Lilli v dostupnej vzdialenosti manipulátora. Pre každý počet objektov (1, .., 4) bolo vykonaných 10 testov, pričom rozmiestnenie kociek a ich natočenie bolo pri každom teste zmenené.

Jednotlivé čísla predstavujú, koľko objektov bolo detegovaných v jednotlivých testoch. Ideálnym prípadom je detekcia takého počtu objektov, koľko ich reálne bolo v scéne. Čiastočným úspechom je, keď detekcia odhalila maximálne o jeden objekt viac, ako bolo v scéne. Neúspechom je detegovanie menšieho počtu objektov alebo väčšieho (aspoň o dva) počtu objektov, ako sa reálne nachádzalo v scéne.

3.1.1 Detegovanie štyroch objektov

Prvým testom bolo detegovanie štyroch objektov, resp. kociek naraz. Výsledky vidíme v priloženej tabuľke:

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
Objekty: 4	3	4	3	4	4	4	5	4	6	6

Čím je povrch členitejší, tým "horšie a nepresnejšie"je vytvorenie mračna bodov. Tento problém nastal, keď sme scénu "rozčlenili"niekoľkými objektmi (konkrétne 4). Pri hranách objektov vznikalo množstvo šumu, ktorý zasahoval aj do samotných objektov a

sťažoval ich detekciu. Pri odstránení šumu došlo k odstráneniu časti objektu, čo následne spôsobilo, že objekt sa rozdrobil na niekoľko častí. Pri volaní funkcie **clusterExtraction()** (2.6.1) následne došlo k nedetegovaniu objektu z dôvodu príliš malých zhlukov bodov v mračne (boli príliš malé, aby boli vyhodnotené ako objekt).

Delenie zhluku bodov na menšie dochádzalo aj v prípade horších svetelných podmienok, kedy sa na hrane lámali objekty na dva a viac (čo nám v kombinácii s väčším počtom objektov skresľovalo výsledky). Najhoršie výsledky sme dosiahli na kockách červenej a žltej farby.

3.1.2 Detegovanie troch objektov

Druhým testom bolo detegovanie troch objektov. Výsledky vidíme v priloženej tabuľke:

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
Objekty: 3	3	3	4	3	5	5	3	4	3	3

Pri testovaní sme narazili na podobné problémy ako pri testovaní štyroch objektov (3.1.1). Keďže sme mali menej objektov, mračno bodov bolo presnejšie a nevznikalo toľko šumu, ako pri štyroch objektoch.

3.1.3 Detegovanie dvoch objektov

Tretím testom bolo detegovanie dvoch objektov. Výsledky vidíme v priloženej tabuľke:

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
Objekty: 2	3	2	2	2	2	2	3	3	2	2

Keďže sme opäť znížil počet objektov, dostávame ešte presnejšie mračno bodov a teda aj samotné objekty sú detegované presnejšie. V testoch sme neodstali ani jeden neuspokojivý výsledok.

3.1.4 Detegovanie jedného objektu

Posledným testom bolo detegovanie jedného objektu. Výsledky vidíme v priloženej tabuľke:

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
Objekty: 1	1	1	1	1	2	1	1	1	1	1

Pri jednom objekte je mračno bodov najčistejšie. Objekt bol presne detegovaný takmer v každom jednom prípade, aj v zhoršených svetelných podmienkach pri rôznych farbách kocky a jej natočeniach.

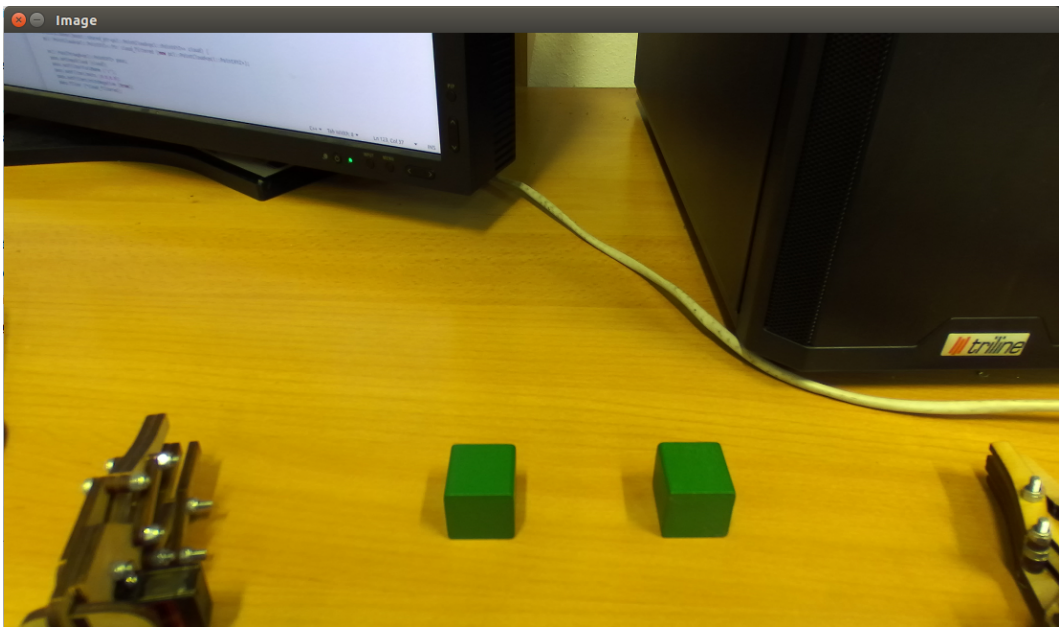
3.1.5 Celkové výsledky

V priloženej tabuľke môžeme vidieť celkové výsledky detekcie objektov vyjadrené percentuálne.

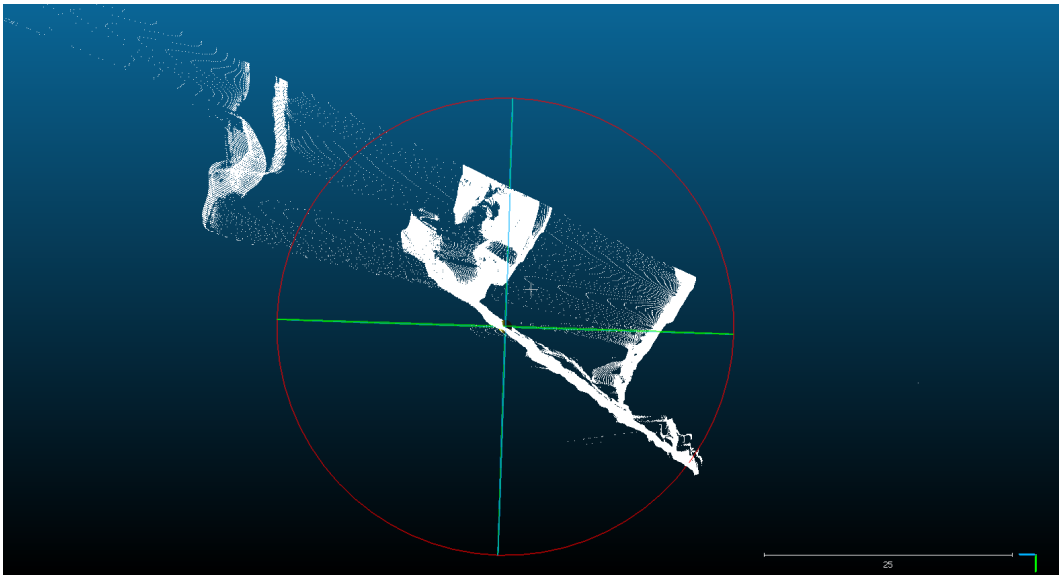
Celkové výsledky	
Počet objektov	Úspešnosť
4	55%
3	70%
2	85%
1	95%

Keďže hlavným cieľom bola detekcia jedného objektu, môžeme považovať výsledky za uspokojivé.

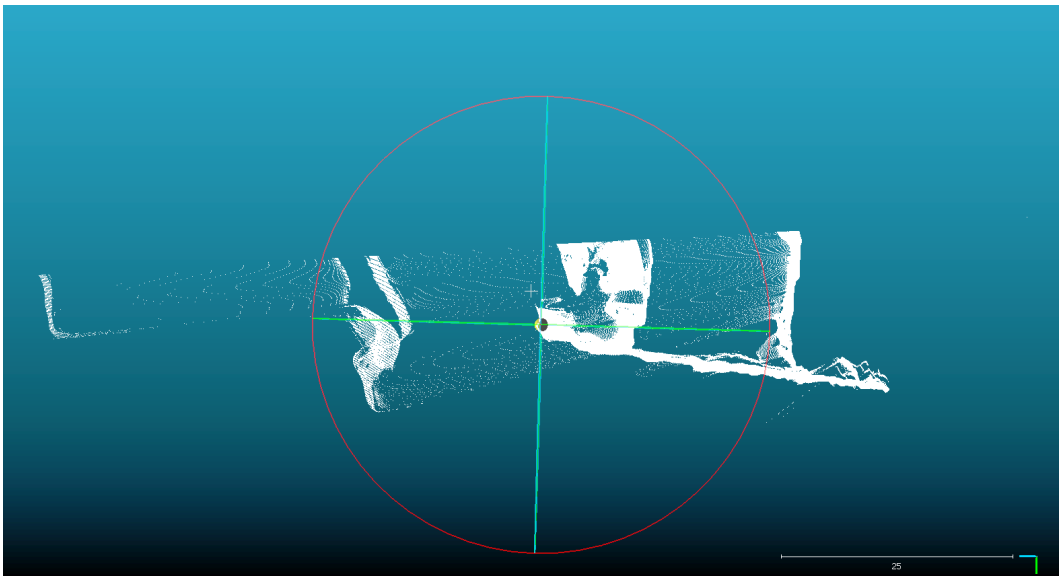
3.1.6 Vizualizácia výsledkov - obrazová dokumentácia



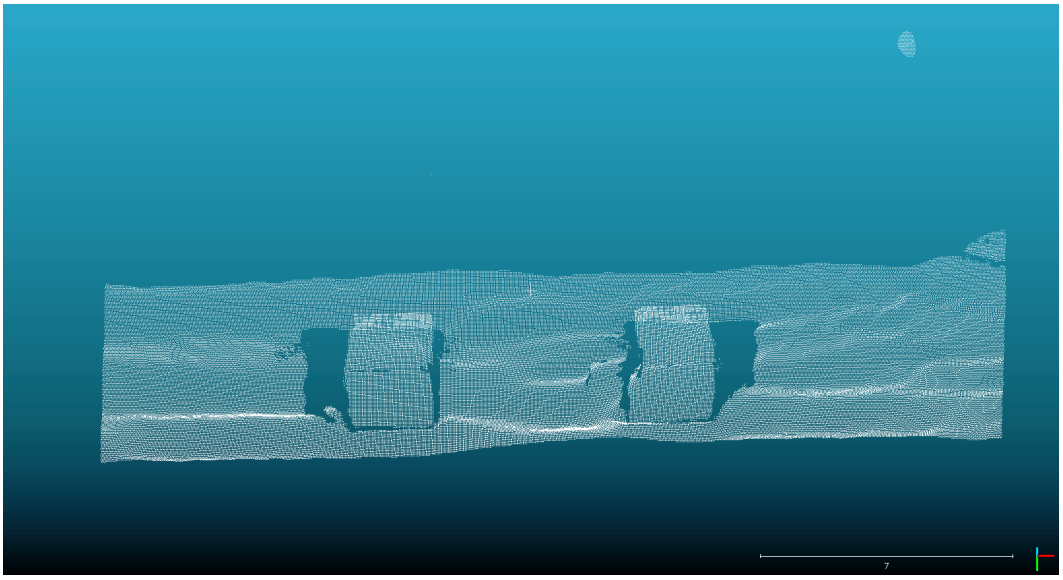
Obr. 3.1: Záber z kamery na scénu



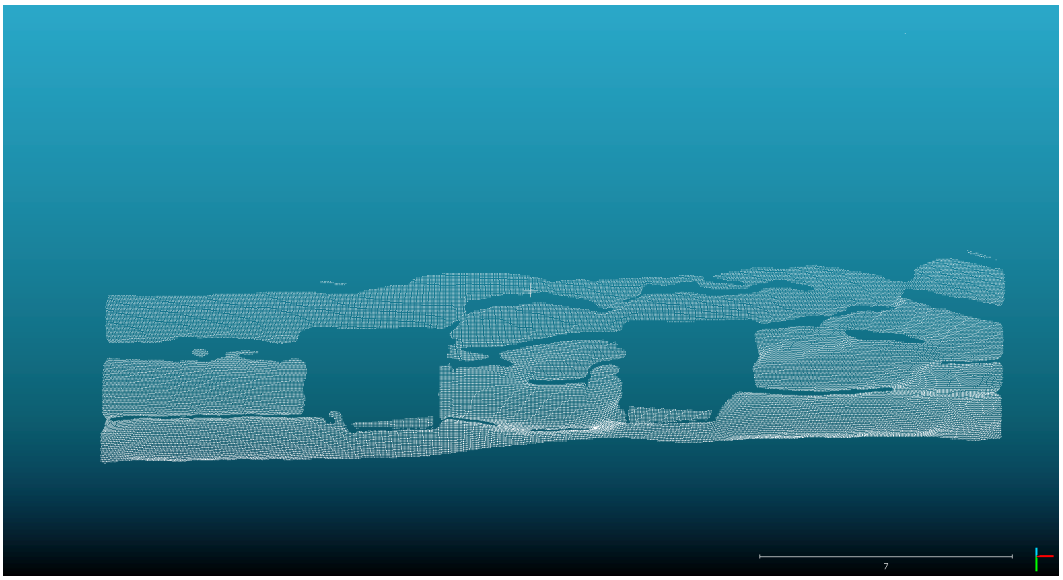
Obr. 3.2: Pôvodné mračno bodov, ktoré je neotočené



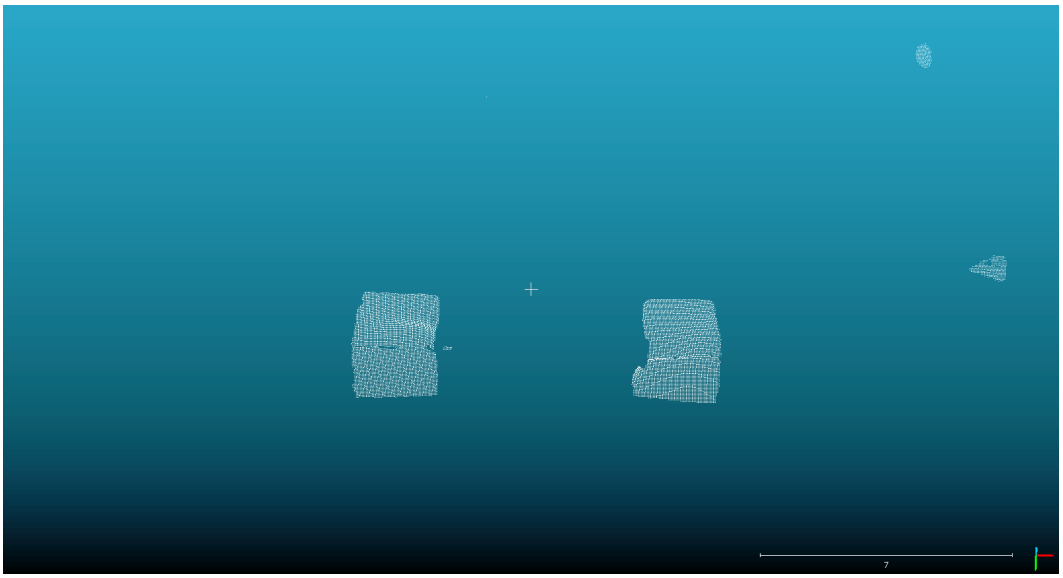
Obr. 3.3: Mračno bodov otočené za pomoci (2.5.1)



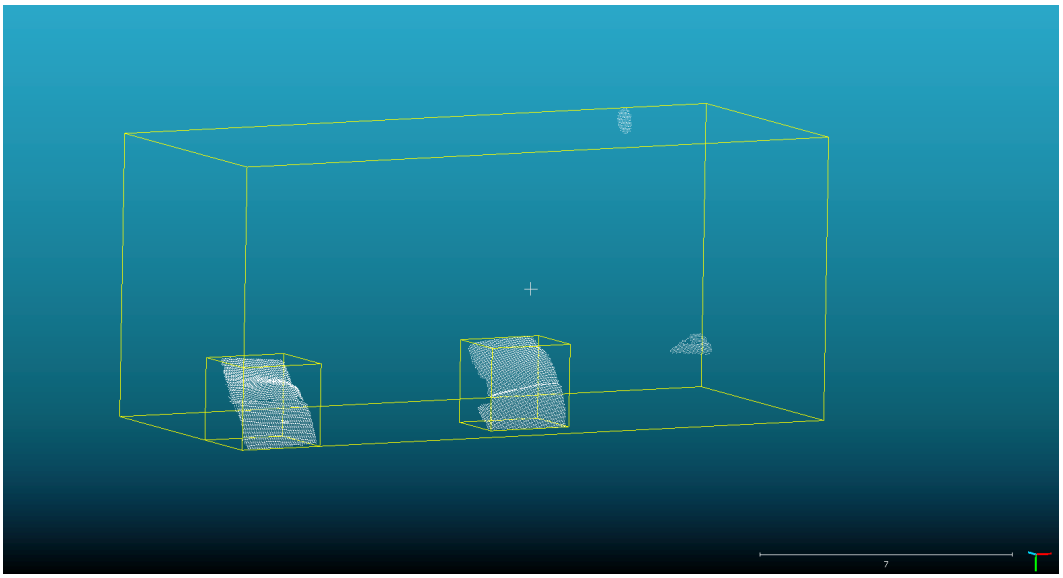
Obr. 3.4: Orezané mračno bodov (2.5.2) - pri porovnaní s predchádzajúcim obrazom môžeme tískať predstavu, o koľko sa mračno zmenšilo



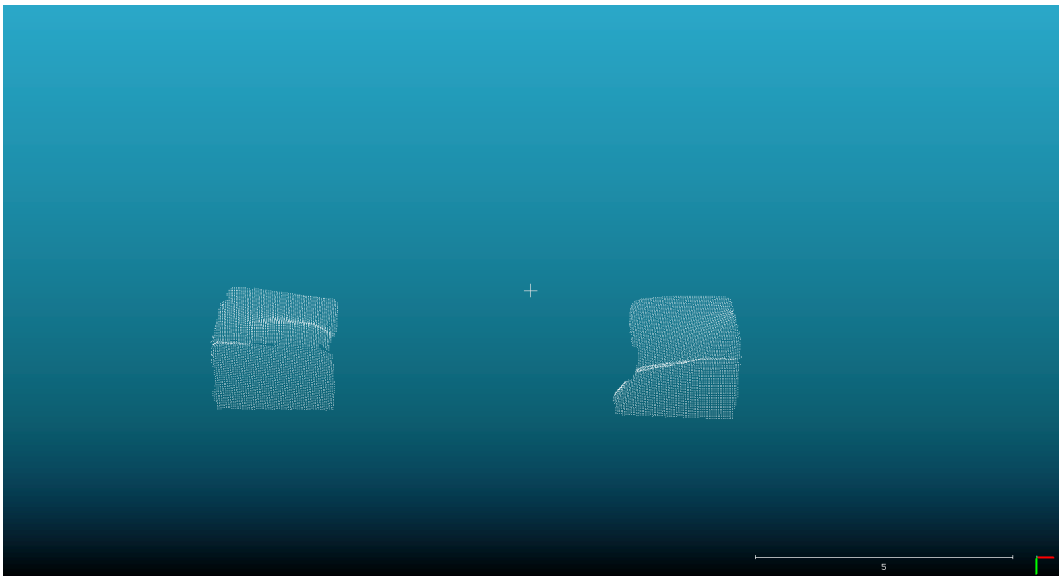
Obr. 3.5: Detegovaná rovina (2.5.4) za použitia algoritmu RANSAC (1.21)



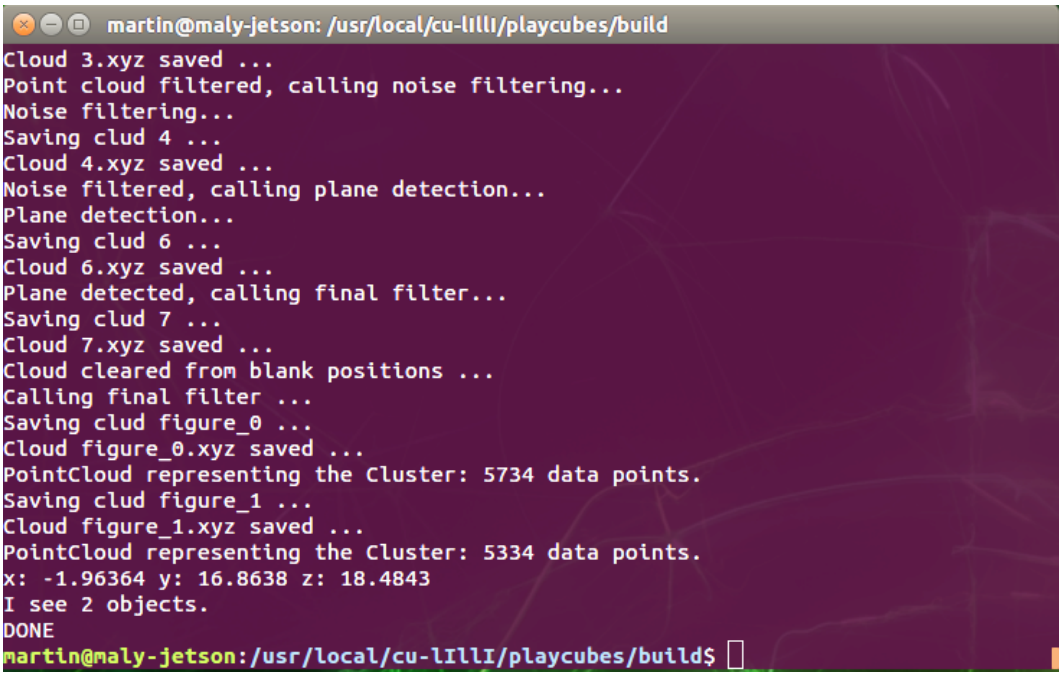
Obr. 3.6: Zobrazenie zostávajúcich zhlukov v mračne bodov po odstránení bodov roviny



Obr. 3.7: Najväčší žltý kváder označuje priestor mračna bodov, dva menšie kvádre označujú detegované objekty



Obr. 3.8: Zobrazenie dvoch detegovaných objektov - tieto sú uložené vo vektoroch, v ďalšom vektore sú uložené stredy objektov, ktoré sa ďalej posúvajú na riešenie inverznej kinematiky



Obr. 3.9: Výpis v konzole - môžeme vidieť počet bodov v jednotlivých zhlukoch objektov, stred prvého z nich a hlásenie o počte detegovaných objektov

Záver

Cieľom tejto práce bolo navrhnúť, implementovať a overiť algoritmy pre pohyb a manipuláciu v prostredí humanoidného robota Lilli. V úvodnej kapitole sme si predstavili úvod do problematiky, popísali problémy a vysvetlili používané algoritmy. V druhej kapitole sme sa venovali návrhu a implementácii nášho riešenia, vysvetlili sme si jednotlivé metódy a ich funkcionalitu. V tretej kapitole sme zhrnuli výsledky rozpoznávania obrazu a ich výslednú úspešnosť.

Hlavným cieľom bola detekcia objektov v scéne pred Lilli. Na zaznamenanie mračna bodov sme využili kameru ZED mini, samotné mračno bodov sme spracovali pomocou Point Cloud Library. Počas detekcie objektov sme však narazili na niekoľko problémov. Hlavným problémom bolo osvetlenie scény a objektov v nej. Rôzne zdroje svetla vytvárali tiene, ktoré nepriaznivo ovplyvňovali detekciu objektu. Mnoho objektov bolo chybne rozdelených na viac menších objektov, čím vznikal väčší počet objektov, ako bol v skutočnosti. Ďalším problémom bolo viac objektov na scéne. S narastajúcim počtom objektov v scéne klesala i presnosť detegovania objektov. Väčšie množstvo objektov znamenalo viac šumu i odrazov svetla. Tieto faktory nepriaznivo vplývajú na kvalitu mračna bodov. Zistili sme, že algoritmus najlepšie funguje s jedným až dvoma objektami tmavších farieb, ktoré nie sú lesklé, teda zlé svetelné podmienky ich až tak neovplyvňujú. Zároveň sme inicializovali inverznú kinematiku ramena a pripravili podmienky pre implementáciu pohybu.

V budúcnosti možno nadviazať na prácu, konkrétne na implementáciu a overenie algoritmu chôdze Lilli a vylepšenie inverznej kinematiky ramena. Taktiež je možné implementovať rozpoznávanie už detegovaných objektov.

Zdrojové súbory sú dostupné na `https://github.com/Robotics-DAI-FMFI-UK/cu-lIllI/tree/master/playcubes`. Všetky zdrojové súbory sú open-source.

Literatúra

[1] Cu lilli. https://kempelen.dai.fmph.uniba.sk/lilli/index.php/CU_Lilli. citované 01.08.2020.

[2] Euclidean cluster extraction. https://pcl.readthedocs.io/projects/tutorials/en/latest/cluster_extraction.html. citované 02.08.2020.

[3] Moveit. https://moveit.ros.org/. citované 05.08.2020.

[4] OpenRAVE documentation - ikfast Module. http://openrave.org/docs/latest_stable/openravepy/ikfast/#ikfast-the-robot-kinematics-compiler. citované 05.08.2020.

[5] Orocos wiki - kinematic trees. https://www.orocos.org/wiki/main-page/kdl-wiki/user-manual/kinematic-trees. citované 06.08.2020.

[6] Orocos wiki - orocos kinematics and dynamics. https://www.orocos.org/kdl. citované 05.08.2020.

[7] ROS wiki - <Joint> element. http://wiki.ros.org/urdf/XML/joint. citované 06.08.2020.

[8] ROS wiki - robot operating system - ROS. http://wiki.ros.org/ROS/Introduction. citované 05.08.2020.

[9] Two-view geometry. http://www.cs.unc.edu/~lazebnik/spring11/lec14_epipolar.pdf. citované 12.05.2020.

[10] Asim Bhatti. *Stereo vision*. BoD–Books on Demand, 2008.

[11] I Blanárik. Epipolárna geometria. *Fakulta informatiky a informačnỳch technoógií, Slovenská technická univerzita v Bratislave*, 2005.

[12] Siyuan Chen, Debra Laefer, and Eleni Mangina. State of technology review of civilian uavs. *Recent Patents on Engineering*, 10:1–1, 07 2016.

[13] Eduardo Bayro Corrochano. *Geometric Computing for Perception Action Systems: Concepts, Algorithms, and Scientific Applications*. Springer Science & Business Media, 2001.

[14] Michal Fikar and Nina Lúčna. Lilli object interaction. `https://wiki.robotika.sk/robowiki/index.php?title=Lilli_object_interaction_-_Michal_Fikar,_Nina_L%C3%BA%C4%8Dna`, 2019. citované 02.02.2020.

[15] Gabriel Halasi. Humanoid robot lilli. Diplomová práca, Univerzita Komenského v Bratislave, Fakulta matematiky, fyziky a informatiky, 2020.

[16] Howard J. Hamilton. Inverse kinematics. `http://www2.cs.uregina.ca/~anima/408/Notes/Kinematics/InverseKinematics.htm`. citované 03.08.2020.

[17] Xian-Feng Han, Jesse Jin, Ming-Jie Wang, Wei Jiang, Lei Gao, and Liping Xiao. A review of algorithms for filtering the 3d point cloud. *Signal Processing: Image Communication*, 57:11, 05 2017.

[18] JERRY HILDENBRAND. Nvidia jetson tx2 is the supercomputer that's going to build the next great idea. `https://www.androidcentral.com/nvidia-jetson-tx2-best-development-package-next-great-idea`, 2017. citované 07.07.2020.

[19] Ichiro Kato, Sadamu Ohteru, Katsuhiko Shirai, Toshiaki Matsushima, Seinosuke Narita, Shigeki Sugano, Tetsunori Kobayashi, and Eizo Fujisawa. The robot musician 'wabot-2'(waseda robot-2). *Robotics*, 3(2):143–155, 1987.

[20] Benio Kibushi, Shota Hagio, Toshio Moritani, and Motoki Kouzaki. Speed-dependent modulation of muscle activity based on muscle synergies during treadmill walking. *Frontiers in Human Neuroscience*, 12, 01 2018.

[21] Marian Mihalik and Mária Zentková. Počítačové videnie v praxi. 2016.

[22] Marek Šolony. *Identifikácia pohybu v priestore*. Bakalárska práca, Vysoké učení technické v Brně, Fakulta informačních technológií, Ústav počítačové grafiky a multimédií, 2007.

[23] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Realtime computer vision with opencv. *Queue*, 10(4):40–56, April 2012.

[24] Nizar Rokbani, Alicia Casals, and Adel M. Alimi. *IK-FA, a New Heuristic Inverse Kinematics Solver Using Firefly Algorithm*, pages 369–395. Springer International Publishing, Cham, 2015.

[25] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[26] Roman Stoklasa. *Segmentácia obrazu použitím hĺbkovej mapy*. Bakalárska práca, Masarykova univerzita, Fakulta informatiky, 2008.

[27] Mgr. Ľudovít Balko PhD. Epipolárna geometria. page 6. citované 07.04.2020.